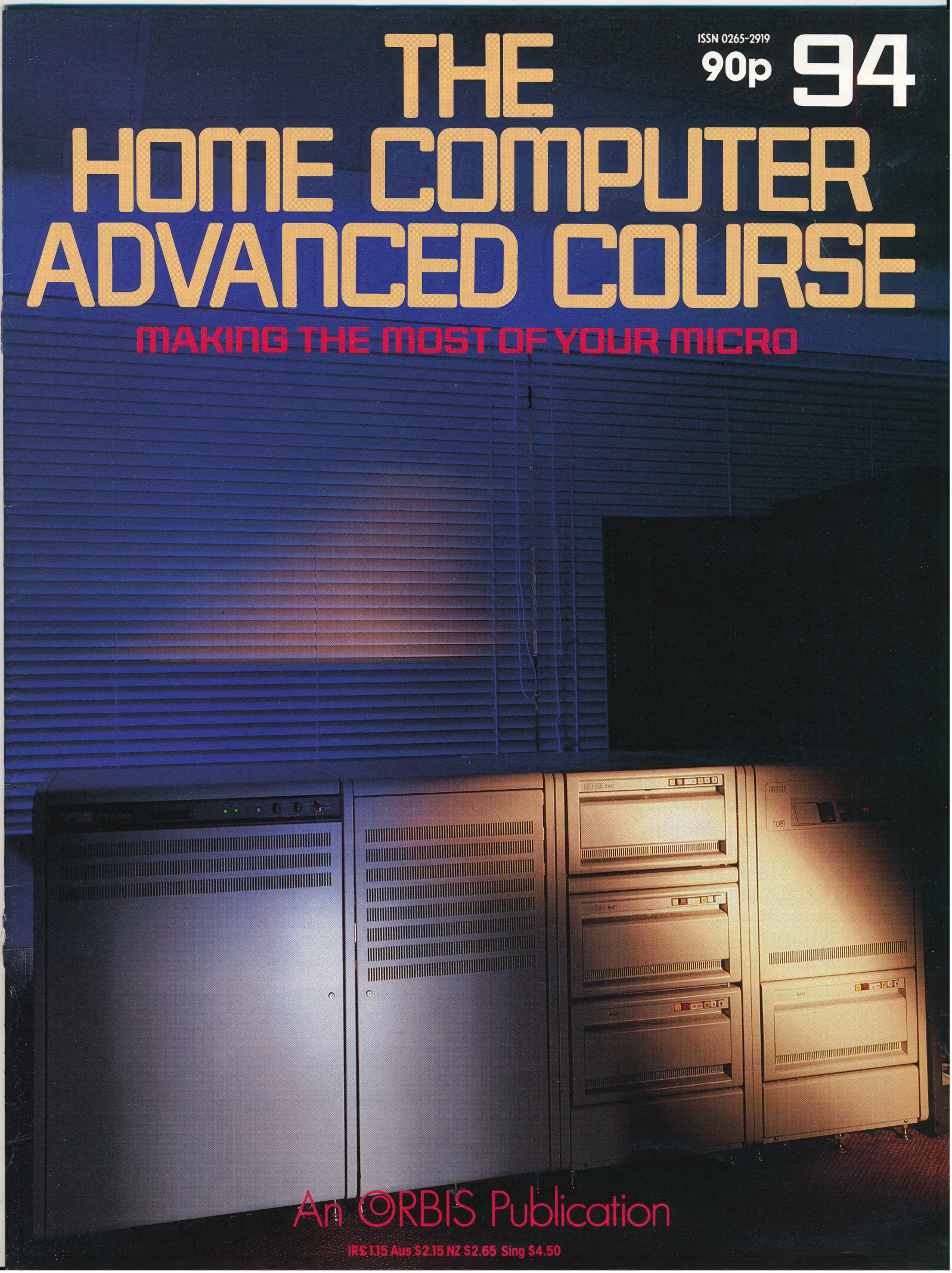
ISSN 0265-2919

90p 94

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 Sing $4.50

CONTENTS

Next Week

- We examine the prospects for the future development of the microcomputer industry.
- We envisage the kind of microcomputer that will be coming off the production lines in the early 1990s.
- It is important for anyone wishing to keep abreast of the rapid changes in the micro industry to read the wealth of literature on the subject. We will be giving examples of some of the books that are available.

QUIZ

1) What is the difference between the VAX Unibus and Massbus?
2) Why are 'pseudo-ops' so called and what is their function?
3) Why do Winchester disks have to be sealed in airtight containers?
4) C contains a number of commands to configure the memory. Why are these commands necessary?

Answers To Last Week's Quiz

1) Yes it does. The fact that a number of processors are housed in the same computer does not alter the essential architecture of each CPU.
2) 'History dependent' means that the current state of a program depends on the results of previous decisions and calculations.
3) Virtual memory management is enjoying a renaissance because many computers are now provided with more memory than their operating systems can cope with.
4) There is no difference at all. C treats information to both files and devices as a stream of bytes.

Issue 96 Index

- The final instalment of the series, issue 96, will be a complete index to the eight volumes of the course. It will also include ready-reference guides to the subjects covered in each section.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Mike Curtis, Steve Malone, David Fensome, Surya, Dave Nicholls, Max Phillips; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE;

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

INSIDE VAX

While the gap between microcomputers and minicomputers is closing rapidly, the sheer power of minis cannot yet be matched by their smaller counterparts. We look here at the DEC VAX 11/780 minicomputer, which offers us some insight into the capabilities of today's powerful development and database machines.

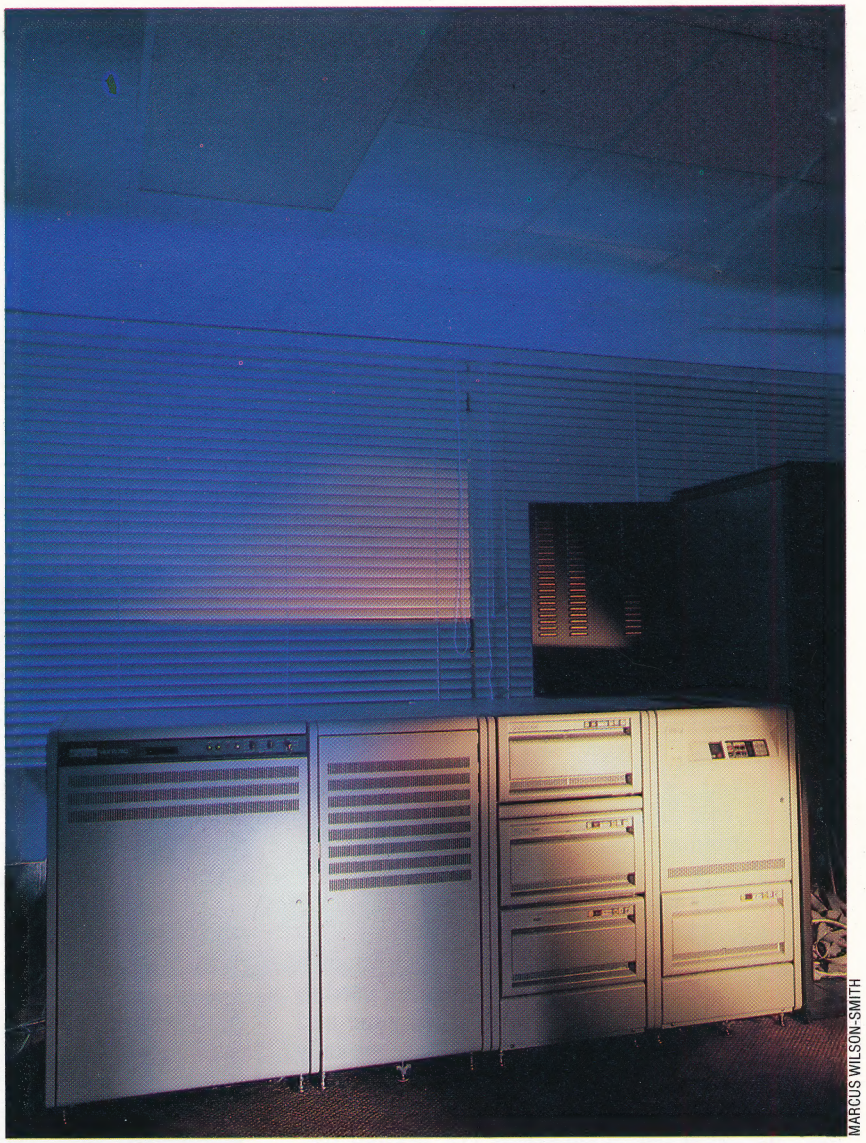
Given the power and speed of today's micros and the advent of networking software and hardware, you could be forgiven for wondering why people still buy large expensive minicomputers from manufacturers like Prime and DEC. Surely the same customers could get by with IBM PCs and a couple of disks each? The main reason for buying a minicomputer, though, is its raw power. The 'power' of a computer can be assessed by considering its computational speed, memory size and the range of software and firmware available. In this respect, none of today's micros even come close to the power of medium-sized minis, and even in a network the computing power is localised — each user is limited to the power of his own machine.

Moreover, if one user is reading a listing, for instance, then his CPU will be idle, preventing anyone else on the network from accessing it. In contrast to this, the entire power of a minicomputer is available to all users, albeit for only part of the time, and if a user doesn't need his 'time slice' then it will be shared among the other users in the system.

One of the most popular minicomputers is the VAX from the Digital Equipment Corporation (DEC). VAX, in fact, comes in several models, ranging from the MicroVax, which in size at least is comparable with a standard business micro, to the top of the range 11/785. We've used the 11/780 as an example, although the architecture is the same for the entire range, with the exception of the 11/782 and 11/785.

Physically, the VAX is rather different from standard home or business micros. For a start, it's much larger — the basic cabinet stands 150cm high, 100cm wide and 75cm deep. Also, whereas most home micros are built onto a single board, with extra boards used only for expansion, an 11/780 can accommodate 16 boards containing only memory chips, though it will always contain several other boards that include interface hardware and the CPU.

A VAX CPU isn't on a single chip, nor even a single board. It's designed to be multi-tasking and to enable efficient handling of 'virtual memory'.



MARCUS WILSON-SMITH

Processing Powerhouse
At £140,000 the DEC VAX 11/780 minicomputer is only a financially viable option for medium and large companies. However, the benefits of mass on-line storage, high speed data processing and the flexibility of this integrated computing facility are generally considered to be worth the large purchase and maintenance costs

The VAX is a 32-bit machine, which gives access to over four gigabytes (four billion bytes) of memory, and even with the falling price of RAM, that much memory would still be very expensive. The virtual memory concept was developed to take advantage of the machine's addressing capabilities. Only a fraction of the memory available is actually RAM; and hardware within the machine is used to check whether the address required is currently in memory; if not, then it is automatically fetched from disk.

This process is called 'paging'. The memory is handled in 512-byte pages, and when the memory is full, pages that have been changed while in memory are written back to disk as new pages are required. This is all completely transparent to the user, however, since all programs 'think' they're talking to a full four gigabytes. In fact, the hardware does all the work automatically.

DEC's standard operating system is VMS (although Unix is available), which has a powerful command language and a comprehensive set of utilities, including, among others, full screen and line editors, sort routines and electronic mail facilities. The only language supplied with the

machine is a macro assembler, but compilers for almost any language are available from DEC or other sources at extra cost. VAX assembly is something of a revelation for programmers used to Z80 or 6502 code. There are, for example, 16 registers, each of which is 32 bits in size. Four of these are special, being used as stack pointers and the program counter, but 12 are for general use. And whereas micros can usually only handle integer arithmetic in hardware, the VAX has instructions for floating point (up to 128-bit floating point numbers) and character handling. There is also a single instruction that corresponds to the BASIC FOR...NEXT loop structure. Included in the full complement of 248 instructions are the basic move and compare operations, and a set of special instructions for CPU control.

To use a VAX you need a user name and password, allocated by the system manager, which allow you to log on. Once on the machine you have access to whatever facilities the system manager has decided you need. Each user name has associated privileges and rights that limit the files you can access, and it also prevents unauthorised users from getting at certain parts of the operating system. Additionally, you can protect your own files so that only certain people can read, write to, delete, or modify them.

The command language is known as DCL (Digital Command Language) and includes many facilities normally found only in high-level languages. These include named variables, GOTO and IF statements similar to their BASIC equivalents, and a set of 'lexical' functions, which are used to return system information such as the time and names of logged on users. Most of these are only really used in 'command procedures', which are basically lists of DCL commands stored in a file and executed like a program. For normal use, however, a single command is used — for example, Show Users will print, naturally enough, a list of users. A lexical function could be used within a command procedure, for instance, to send each one a message.

Most of the commands have several qualifiers that are used to modify the action. As an example, DELETE *.* will delete all files in a directory (subject to the files not being protected) and DELETE/CONFIRM *.* will perform the same action, but you'll be prompted at each file for confirmation. The full command list is extensive, and includes all the usual file-handling instructions, as well as some to modify the working environment and to allow you to communicate with other users.

Communicating with the machine is done through two special interfaces. The Unibus is used for terminals, slow disks, tape drives, printers, and a variety of special-purpose equipment like plotters and control devices (which are available). The other interface is the Massbus, which is much faster and is used for fast-access disk drives.

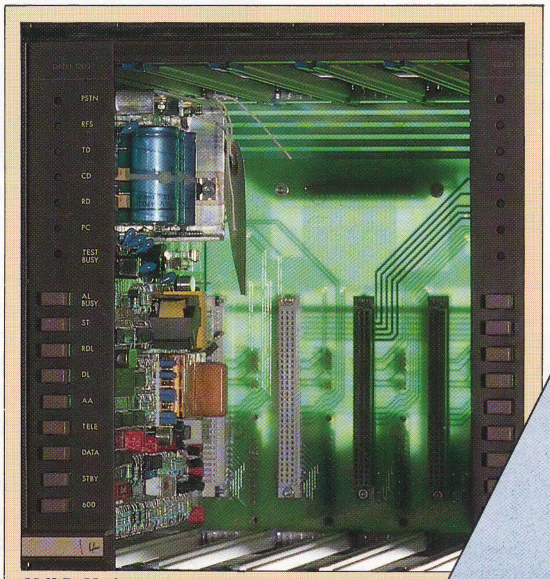
The smallest configuration of an 11/780 has two megabytes of memory and eight terminal lines. You have to buy at least two disk drives, each of which must be capable of storing at least two megabytes in a workable small system. Up to eight disk drives can be fitted, and the memory can be expanded to eight megabytes. Up to 64 terminals can be connected by simply installing extra interface boards.

As a development environment, VMS is excellent: the command language is logical, and there are online help files for all commands and command qualifiers. Libraries of useful routines are also provided for use in your own programs. This means that development of things like screen handling and special-purpose maths routines is eliminated, resulting in large reductions in programming time. Disk file sharing by a development team eliminates the problems accruing from stand-alone micro systems when someone changes a file without telling other users to change theirs.

The minicomputer provides a very friendly programming environment, and the power of the machine allows complicated applications programs to run more quickly than on a micro – but power has its price. A standard VAX 11/780 with a pair of 205 megabyte disk drives will cost at least £140,000. Minicomputers also generate high running costs; in maintenance, electricity supply (about 6kW minimum) and air conditioning. As micros get more powerful though, minicomputers will eventually be replaced, but for the time being they provide users with performance second only to mainframes.

The American Console
The VAX's system manager communicates with the machine via the operator's console (shown right), using it much like a tele-typewriter. The computer can also send responses and status messages back to the console, where they are printed. Thus, a complete hard copy of any dialogue between the manager and machine is produced

The Line Printer
A high-speed line printer (shown right) can be connected to the VAX via the Unibus



MARCUS WILSON-SMITH

M.U.D. Modems
British Telecom's Multi -User Dungeon is an interactive adventure game with over 1,000 locations that users phone on modem-equipped micros to play. The game is run during non-business hours on a VAX minicomputer, which during the day acts as a European database. The VAX is equipped with numerous modems (as shown above) attached to its Unibus network, allowing up to 100 people to play M.U.D. simultaneously, each on their own private telephone link

KEVIN JONES

Mini System
The VAX communicates with its peripherals via two buses. Massbus is a high speed bus that allows fast data transfer between disk or high-speed tape machines and the computer. Unibus is used for slower peripherals. Eight terminals can be attached, as well as printers and plotters. Unibus adds flexibility to the VAX system, since it allows communications links and industrial process control devices to be linked to it

DEC VAX 11/780

PRICE
£140,000 standard. A pair of 205 megabyte disk units costs an additional £35,000. Terminals are £800 each

DIMENSIONS
1,181×762×1,537mm

CPU
VAX CPU

MEMORY
Between 2 and 8 megabytes RAM. Virtual memory management allows access to 4 gigabytes

SCREEN
At least 80×25 text display and up to 1,024×1,024 pixel resolution with 16.5 million colours (dependent on terminal type)

INTERFACES
Massbus for fast disks and tapes, Unibus for all other devices including printers, plotters and terminals, via an RS232 link

LANGUAGES AVAILABLE
Supplied with assembler kit, but compilers for most high-level languages are available at extra cost

DOCUMENTATION
The basic set of documentation for the command language and utilities runs to 30 volumes, occupying 5 feet of shelf space. This includes a concise reference section and tutorial guides.

STRENGTHS
The VAX provides an extremely powerful programming environment for software development and database management. Networking between VAX machines is also possible

WEAKNESSES
The system is extremely expensive and incurs high running costs, since it requires a dust-free atmosphere and full-time maintenance and support

WRITING YOUR OWN SCRIPT

By combining the abilities of the Unix shell with the operating system's redirection and piping instructions, users are able to rewrite the system to their own specifications. We look at some of these, and present an example that amply illustrates Unix's adaptability to its users' requirements.

Most operating systems include facilities that allow users to customise the system to meet their specific requirements. These usually include the options of running a batch of commands at once (the CP/M SUBMIT and the MS-DOS .BAT files, for instance), and running a particular sequence of commands or programs when the user first starts up the system. Unix, however, goes further.

Unix has a 'shell script', a programming language in its own right, which has facilities for input/output, selections and iterations. Combined with the Unix redirection and piping operators, this enables skilled users to construct programs for most tasks without the need for a 'conventional' programming language.

The shell is the equivalent of the command processor in CP/M and it can be completely rewritten by the user. There is a slight disadvantage in that different Unix systems may present different 'faces' to the user, although the underlying system is identical. There are two common shells for the Berkeley Unix system, on which this series is based. The 'C shell' as its name suggests, is based on the C language, and is what we've used in our examples. Had we used the 'Bourne shell', however, there would have been very few differences in the things we have considered so far.

Let's look at some of the commands that affect the user's environment. We have already seen briefly how each file or directory can have access privileges for three groups of people: the users themselves (u), the user's group (g) and others (o). (This idea of groups is very useful, especially in large organisations where the pooling of resources and the assessment of each pocket of work is vital.) Each of the three categories is given access rights for every file and directory on the system, denoted by three sets of rwx in the full directory listing given by the ls -l command. The r denotes access to read, w to write and x to execute. If a category does not have the appropriate access, the letter is replaced by a dash.

These access rights can be changed using the chmod command, which takes the form:

```
chmod category[+,-]access fileordirectoryname
```

where + is used to give an access right, and - is used to remove one. So, to give the user and their group write access to a file called partwork, the following would be acceptable:

```
chmod ug+w partwork
```

Note that more than one permission change can be given, separated by commas (but not spaces).

Another option allows you to change the settings for the particular terminal type being used. Most Unix systems come with built-in tables for a variety of terminal types, one of which can be selected by the use of:

```
setenv TERM terminaltype
```

This means that software packages don't have to be installed for a particular screen and keyboard; in theory, one installation will work for all of them.

For non-standard terminal types and for particular purposes, the keys used for various functions can be changed using the stty command:

```
stty everything
```

will give a list of all the settings available, and:

```
stty all
```

will give a restricted list of all the common ones.

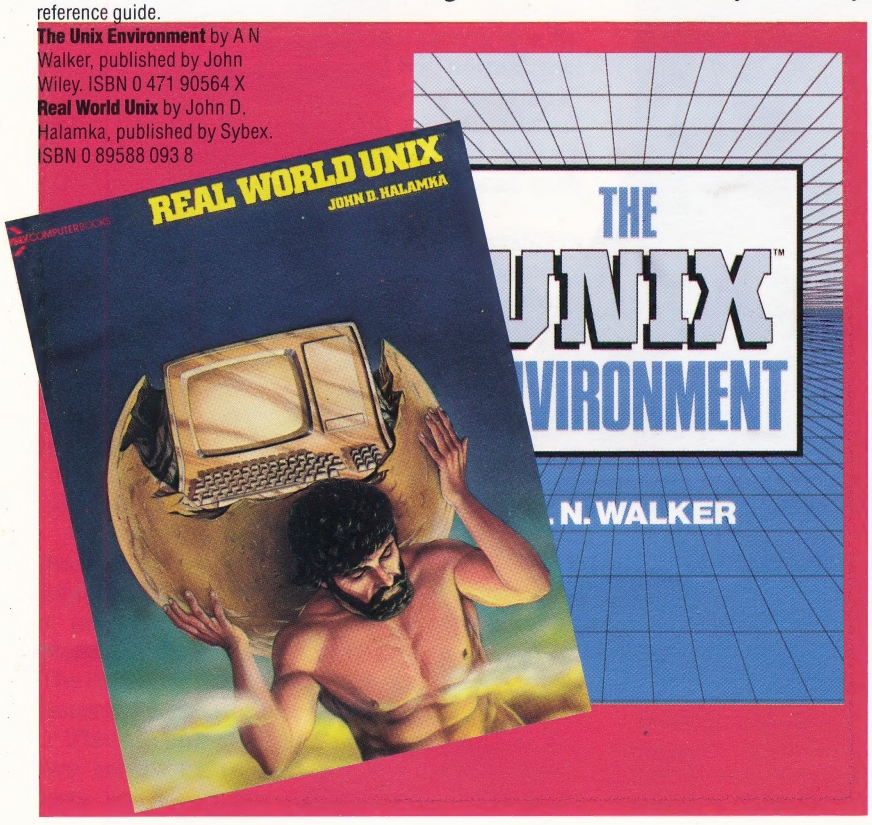
Each function has a name and can be changed:

Unix Texts

Many books about Unix are available. These range from thick 'reference' works for the expert programmer to simple introductory guides for the casual user. The Unix Environment by A N Walker is written in an entertaining style, and is suitable for readers new to Unix. It covers the main features of the operating system, as well as discussing the philosophy behind Unix, its typical hardware facilities and system management.

John Halamka's Real World Unix forms an excellent practical text for those who want to learn Unix in the shortest possible time. The text contains numerous examples of the most common tasks and a 'one minute Unix' quick reference guide.

The Unix Environment by A N Walker, published by John Wiley. ISBN 0 471 90564 X
Real World Unix by John D. Halamka, published by Sybex. ISBN 0 89588 093 8



Custom Job

```
Berkeley 4.2 Vax/Unix (infsc3)
Type (Ctrl-D) to disconnect

login: com-mcc
Password:
You are a Normal user (class 3)
Jobs : 15  Superiors : 4  Maximum : 21
Last Login: Wed Nov 13 15:05:36 on ttyn08

Hello nice to see you!          (this is my user login message)

%cat .login                     (look in .login file)
setenv TERM tvi910+             (set to my type of terminal)
stty erase                      (Ctrl-H enables me to use backspace
                                 on my PC instead of 'del' key)
echo Hello nice to see you!     (this is my sign on message)

%cat .cshrc
setenv EDITOR /usr/local/emacs
setenv NAME 'Mike Curtis'
set path=(. /usr/local/m2 /usr/local /usr/ucb /usr/bin /bin /usr/games )
(the path sets the directories that will be searched (in order
given) for commands, so programs in other directories can be
accessed directly if required. The first '.' is user's own
directory)

set history=25                  (shell will remember last 25
                                 commands)
set ignnoreeof                  (stops accidental logout with Ctrl-D)
alias h history                 (customise Unix commands)
alias ty more
alias dir ls                    (use dir instead of ls to list
                                 directory)

%dir                            (demonstrate dir command)
lsfile mike rec.c receive rx.p transmit

%cat > ! temp                   (cat without an input file will take
                                 input from keyboard up to a Ctrl-D)

echo this is a list of my files
/bin/ls                         (the full name of ls command)
%mv temp ls                     (make own customised ls command)

%chmod u+x ls                   (use chmod to make it executable.
                                 Otherwise executed by sh < ls)
%ls                             (try out customised version)
this is a list of my files
ls lsfile mike rec.c receive rx.p transmit

%history                        (review commands issued)
     1  dir
     2  cat > ! temp
     3  mv temp ls
     4  chmod u+x ls
     5  ls
     6  history

% !5                            (repeat command 5)
ls                              (echoes repeated command first)
this is a list of my files
ls lsfile mike rec.c rewceive rx.p transmit

%stty everything                (look at all terminal settings)
new tty, speed 1200 baud
even odd -raw -nl echo -lcase -tandem -tabs -cbreak ff1
-crtbs -crterase -crtkill -ctlecho -prterase -tostop
-tilde  -flusho -mdmbuf -litout -nohang
-pendin -decctlq -noflsh
erase kill werase rprnt flush lnext susp intr quit stop eof
^H    ^U   ^W     ^R    ^O    ^V    ^Z/^Y ^C   ^\   ^S/^Q  ^D

%stty all                       (some common key settings)
new tty, speed 1200 baud; -tabs ff1

erase kill werase rprnt flush lnext susp intr quit stop eof
^H    ^U   ^W     ^R    ^O    ^V    ^Z/^Y ^C   ^\   ^S/^Q  ^D

%logout
```

```
stty functionname newkey
```

Another useful facility prevents unauthorised use of the terminal by the use of lock. This command will ask for a password (different from the login password), and then ask for it to be repeated. The terminal will then be locked and Unix will ignore all key depressions until the password is entered a third time.

The following two features are peculiar to the C shell, but are both interesting and instructive. The history command will give a list of all previous commands issued, up to a predefined number. Additionally, any of the commands can be repeated using the ! operator, usually followed by the number in the list of the command you wanted. Thus:

```
!4
```

would cause the fourth command in the list to be repeated.

The alias command can be used to assign another name to either any command or any string of characters. Wherever the new name is used, Unix will substitute the string in its place. CP/M or MS-DOS users who want to use dir to get a directory listing rather than Unix's ls command:

```
alias dir ls
```

will accept dir as ls, which can then be followed by any of the usual options. If the string contains a semicolon, which Unix uses to separate commands on one line, it can be enclosed in quotes:

```
alias dir 'date;ls -a'
```

Judicious use of alias can alleviate many of Unix's unfriendly features.

There are a number of special system files in most users' directories that are distinguished by having names beginning with a full stop. These are protected against accidental deletion by wild card characters not matching a leading dot. Two of these files are of particular interest as they set up the environment for a user — .login and .cshrc (in the Bourne shell, their functions are covered by a single file, .profile) — and each contains a number of shell commands. The .login file is run whenever the user logs in, and the .cshrc is referenced whenever the shell program is activated (these files usually contain mainly alias, setenv and set commands). The possible options for setenv and set depend on the individual installation and are far too numerous to detail here. But suffice it to say that virtually any aspect of the system and peripheral devices can be set up according to your own specifications.

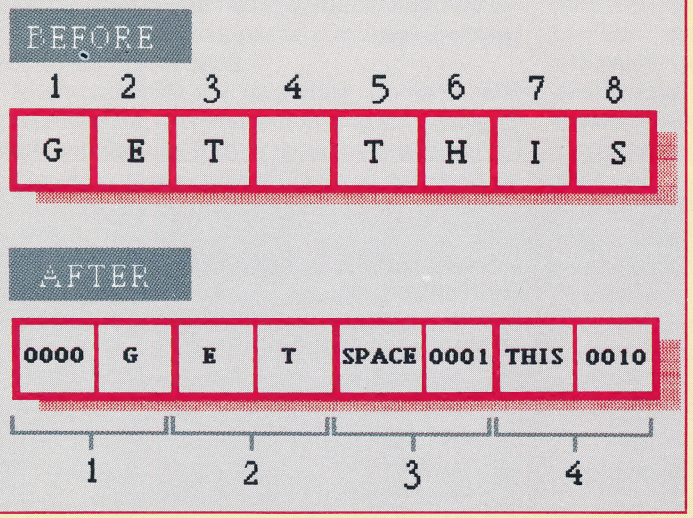
THE SQUEEZE

In this penultimate instalment of our series, we present the 6502 assembly compression routine and a BASIC Loader program for the Z80 version given on page 1855. The second half of the 6502 program, and the BASIC Loader and Driver programs will be printed in the final instalment.

6502 Compression Routine

```
;++++ 6502 TEXT COMPRESSION ++++
;
ZPTR1=$8B               ;0 PAGE INPUT POINTER
ZPTR2=$8D               ;0 APGE OUTPUT POINTER
ZPTR3=$FB               ;UTILITY TABLE POINTER
ZPTR4=$FE               ;UTILITY TABLE POINTER
*=$C000
OUTPUT  *=*+2           ;START OF O/P STRING
INPUT   *=*+2           ;START OF I/P STRING
STATUS  *=*+1           ;STATUS BYTE
MASK    *=*+1           ;NIBBLE MASK
LEN     *=*+1           ;INPUT STRING LEN STORE
OUTOFF  *=*+1           ;OUTPUT OFFSET STORE
TOKCNT  *=*+1           ;TOKEN COUNTER
TABCNT  *=*+1           ;TABLE COUNTER
TEMP    *=*+1           ;SAVE START OF O/P STRING
;
START   LDA INPUT
        STA ZPTR1       ;SET UP I/P POINTER
        LDA INPUT+1
        STA ZPTR1+1
        LDA OUTPUT
        STA ZPTR2       ;SET UP O/P POINTER
        LDA OUTPUT+1
        STA ZPTR2+1
        LDA #1
        STA OUTOFF      ;INIT O/P OFFSET
        LDY #0          ;INIT I/P OFFSET
        LDA (ZPTR1),Y
        STA LEN         ;STORE LENGTH I/P STRING
        LDA #255
        STA MASK        ;SET UP NIBBLE MASK
NCHAR   INY
        LDA (ZPTR1),Y   ;GET NEXT I/P CHAR
        JSR CHECK       ;VALID?
        BNE BADCHR
        JSR TOKEN       ;TOKEN?
        BEQ GOTTOK
        JSR FOURBT      ;FOUR BIT CODE?
        BEQ GOTFOR
        JSR EIGHTB      ;EIGHT BIT CODE?
        PHA             ;SAVE VALUE
        LDA #0
        JSR WRTNIB      ;SEND 0000 CODE
        PLA
GOTFOR  JSR WRTNIB      ;SEND 8 BIT CODE
;
REJOIN  CPY LEN         ;MORE I/P CHARS?
        BCC NCHAR       ;GO GET THEM
        LDA #0
        STA STATUS      ;0 STATUS=OK
        LDA #2
        JSR WRTNIB      ;END OF TEXT MARK
        LDA MASK
        BEQ NODEC
        DEC OUTOFF
NODEC   LDY #0
        LDA OUTOFF
        STA (ZPTR2),Y   ;WRITE O/P LENGTH
        RTS
;
BADCHR  LDA #255
        STA STATUS      ;SEND BAD STATUS
        RTS
GOTTOK  PHA
        LDA #1
        JSR WRTNIB      ;SEND 0001 CODE
        PLA
        JSR WRTNIB      ;AND TOKEN CODE
        JMP REJOIN
;
;++++ SUBROUTINES +++++
TOKEN   PHA             ;SAVE CURRENT CHAR
        TYA
        PHA             ;AND I/P OFFSET
        STY TEMP
        LDA ZPTR1
        CLC
        ADC TEMP        ;CALC ZPTR3 TO
        STA ZPTR3
        LDA ZPTR1+1     ;POINT AT CURRENT
        ADC #0          ;I/P CHAR
        STA ZPTR3+1
        LDA #<TOKTAB
        STA ZPTR4       ;SET ZPTR4
        LDA #>TOKTAB    ;TO POINT TO
        STA ZPTR4+1     ;TOKEN TABLE
        LDA #0
        STA TOKCNT      ;INIT TOKEN COUNTER
TOK1    LDY #0
        LDA (ZPTR4),Y   ;GET LENGTH NEXT TOKEN
        PHA             ;STORE IT
        TAX
        BEQ NOTFND      ;END OF TABLE
        LDA ZPTR4
        CLC
        ADC #1          ;INC ZPTR4
        STA ZPTR4       ;ZPTR3 & ZPTR4
        LDA ZPTR4+1     ;NOW BOTH AT
        ADC #0          ;TEXT STARTS
        STA ZPTR4+1
TOK2    LDA (ZPTR3),Y
        CMP (ZPTR4),Y   ;COMPARE
        BNE NEXTOK      ;NO MATCH GET NEXT TOKEN
        INY
        DEX             ;CONT TILL
        BNE TOK2        ;END TOKEN WORD
;++ TOKEN MATCHED ++
        PLA             ;CLEAR STACK
        PLA
        DEY
        STY TEMP
        CLC
        ADC TEMP        ;CALC NEW I/P OFFSET
        TAY             ;PUT IT IN Y
        PLA
        LDA TOKCNT
        LDX #0          ;SET Z=1
        RTS
;
NEXTOK  PLA             ;GET LENGTH TOKEN
        STA TEMP
        LDA ZPTR4
        CLC
        ADC TEMP
        STA ZPTR4       ;ADJUST ZPTR4
        LDA ZPTR4+1     ;TO POINT AT
```

Slim Lines
The use of tokens, plus the representation of common characters in a four-bit code, enables our text compression algorithm to achieve compression ratios as high as 50 per cent. Here, a string of eight letters (including the space) is reduced to four bytes. Note the 4-bit control codes that signal a tokenised word, a second character set and the end of the message




```
        ADC #0          ;NEXT TOKEN
        STA ZPTR4+1
        INC TOKCNT
        JMP TOK1
NOTFND  PLA             ;DISCARD
        PLA
        TAY             ;RESTORE I/P POINTER
        PLA             ;RESTORE I/P CHAR
        LDX #255        ;Z=0=FAILURE
        RTS
;
FOURBT  PHA             ;SAVE I/P CHAR
        STY TEMP        ;SAVE I/P OFFSET
        LDA #<TAB4BT
        STA ZPTR3
        LDA #>TAB4BT
        STA ZPTR3+1
        PLA             ;GET I/P CHAR BACK
        JMP TBSCAN
EIGHTB  PHA             ;SAVE I/P CHAR
        STY TEMP        ;SAVE I/P OFFSET
        LDA #<TAB8BT
        STA ZPTR3
        LDA #>TAB8BT
        STA ZPTR3+1
        PLA             ;GET I/P CHAR BACK
;
TBSCAN  LDY #0
TBSCN2  CMP (ZPTR3),Y
        BEQ TBSCN1      ;FOUND IT!
        INY
        CPY #16         ;LOOK FOR TAB END
        BNE TBSCN2
        LDY TEMP        ;RESTORE I/P OFFSET
        LDX #255        ;Z=0=FAILURE
        RTS
TBSCN1  TYA             ;PUT TABVAL IN A
        LDY TEMP
        LDX #0          ;Z=1=SUCCESS
        RTS
;
CHECK   CMP #' '
        BEQ RETURN
        CMP #','
        BEQ RETURN
        CMP #'.'
        BEQ RETURN
        CMP #'A'
        BCC NOGOOD
        CMP #$5B
        BCS NOGOOD
RETURN  LDX #0
        RTS
NOGOOD  LDX #255
        RTS
;
WRTNIB  PHA             ;SAVE NIBBLE
        STY TEMP        ;SAVE I/P OFFSET
        LDY OUTOFF
        LDA MASK
        BNE LEFT        ;<>0 MEANS LEFT-HAND
        PLA
        ORA (ZPTR2),Y   ;ADD NEW TO OLD
        STA (ZPTR2),Y   ;REPLACE IT
        INC OUTOFF
        LDY TEMP        ;RESTORE I/P OFFSET
        LDA #255
        STA MASK        ;RESET MASK FOR NEXT
        RTS
LEFT    PLA
        ASL A
        ASL A
        ASL A
        ASL A
        STA (ZPTR2),Y
        LDY TEMP
        LDA #0
        STA MASK
        RTS
;
;++++ MOST COMMON LETTERS ++++
TAB4BT  .BYT 0
        .BYT 0
        .BYT 0
        .BYT 'F'
        .BYT 'L'
        .BYT 'D'
        .BYT 'H'
        .BYT 'S'
        .BYT 'I'
        .BYT 'R'
        .BYT 'N'
        .BYT 'O'
        .BYT 'A'
        .BYT 'T'
        .BYT 'E'
        .BYT ' '
;++++ LESS COMMON LETTERS ++++
TAB8BT  .BYT 'C'
        .BYT 'M'
        .BYT 'U'
        .BYT 'G'
        .BYT 'Y'
        .BYT 'P'
        .BYT 'W'
        .BYT 'B'
        .BYT 'V'
        .BYT 'K'
        .BYT 'X'
        .BYT 'J'
        .BYT 'Q'
        .BYT 'Z'
        .BYT ','
        .BYT '.'
;++++ TOKEN TABLE ++++
TOKTAB  .BYT 3,'THE'

        .BYT 4,'THIS'

        .BYT 4,'THAT'

        .BYT 2,'IF'
```

```
.BYT 3,'YOU'
.BYT 2,'ME'
.BYT 3,'WAS'
.BYT 2,'HE'
.BYT 3,'SHE'
.BYT 4,'THEY'
.BYT 2,'OF'
.BYT 2,'IT'
.BYT 2,'IS'
.BYT 3,'FOR'
.BYT 2,'ON'
.BYT 2,'TO'
.BYT 0
.END
```

Z80 BASIC Loader

```
10 MEMORY 29999
20 FOR n=30000 TO 30640
30 READ p: POKE n,p: c=(c+p)
40 NEXT n
50 IF c<>64282 THEN PRINT "
Checksum error..."
60 STOP
100 DATA 24,7,70,160,102,158,0,255,0,42,50
,117,126,50,56,117
110 DATA 35,237,91,52,117,213,19,237,83,52
,117,62,255,50,55,117
120 DATA 126,205,253,117,32,57,205,162,117
,40,59,205,223,117,40
130 DATA 10,205,232,117,245,62,0,205,22,
118,241,205,22,118,35
140 DATA 58,56,117,61,50,56,117,167,32,216
,50,54,117,62,2,205
150 DATA 22,118,235,209,58,55,117,167,40,1
,43,167,237,82,125,18
160 DATA 201,209,62,255,50,54,117,201,245,
62,1,205,22,118,241
170 DATA 205,22,118,24,203,229,235,33,2,
119,58,56,117,79,6,15
180 DATA 126,167,40,42,213,229,35,197,71,
26,190,32,21,35,19,5
190 DATA 32,13,121,50,56,117,193,225,225,
225,235,43,175,120,201
200 DATA 13,32,231,193,5,225,126,95,22,0,
35,25,209,24,210,225
210 DATA 126,183,201,229,33,226,118,205,
241,117,225,201,229,33
220 DATA 242,118,205,241,117,225,201,6,15,
190,40,5,35,16,250,183
230 DATA 201,120,201,254,32,200,254,44,200
,254,46,200,254,65,56
240 DATA 8,254,91,48,4,79,175,121,201,62,
255,167,201,79,58,55
250 DATA 117,237,91,52,117,167,32,14,26,
177,18,19,237,83,52,117
260 DATA 62,255,50,55,117,201,121,203,39,
203,39,203,39,203,39
270 DATA 18,175,50,55,117,201,42,50,117,35
,34,50,117,237,91,52
280 DATA 117,213,62,255,50,55,117,205,185,
118,254,2,202,129,118
290 DATA 210,153,118,167,202,139,118,205,
185,118,205,110,118,71
300 DATA 126,35,205,174,118,16,249,24,225,
33,2,119,71,62,15,144
310 DATA 71,4,126,35,5,200,95,22,0,25,24,
246,42,52,117,209,167
320 DATA 237,82,125,18,201,205,185,118,33,
242,118,205,164,118,205
330 DATA 174,118,24,182,33,226,118,205,164
,118,205,174,118,24,171
340 DATA 95,62,15,147,95,22,0,25,126,201,
237,91,52,117,19,18
350 DATA 237,83,52,117,201,58,55,117,42,50
,117,167,126,32,14,230
360 DATA 15,35,34,50,117,79,62,255,50,55,
117,121,201,203,47,203
370 DATA 47,203,47,203,47,230,15,79,175,50
,55,117,121,201,32,69
380 DATA 84,65,79,78,82,73,83,72,68,76,70,
0,0,0,67,77,85,71
390 DATA 89,80,87,66,86,75,88,74,81,90,44,
46,3,84,72,69,4,84
400 DATA 72,73,83,4,84,72,65,84,2,73,70,3,
89,79,85,2,77,69
410 DATA 3,87,65,83,2,72,69,3,83,72,69,4,
84,72,69,89,2,79,70
420 DATA 2,73,84,2,73,83,3,70,79,82,2,79,
78,2,84,79,0,205,167
430 DATA 126,32,14,230,15,35,34,50,117,79,
62,255,50,55,117,121
440 DATA 201,203,47,203,47,203,47,203,47,
230,15,79,175,50,55,117
450 DATA 121,201,32,69,84,65,79,78,82,73,
83,72,68,76,70,0,0
460 DATA 0,67,77,85,71,89,80,87,66,86,75,
88,74,81,90,44,46,3
470 DATA 84,72,69,4,84,72,73,83,4,84,72,65
,84,2,73,70,3,89
480 DATA 79,85,2,77,69,3,87,65,83,2,72,69,
3,83,72,69,4,84,72
490 DATA 69,89,2,79,70,2,73,0,0
```

Basic Flavours

Spectrum owners will need to make a small amendment to the Loader program. The compression routine should also work on other Z80 machines, including the Memotech, Amstrad and Einstein. Users of some of these machines may also need to alter line 10 of the Loader program (consult you manual for details):

```
10 CLEAR 29999
```

VLSI

An acronym for 'very large scale integration', the term refers to integrated circuits that contain over 100,000 transistors. The development of *VLSI* chips has been the driving force behind the expansion of 16-bit technology in recent years. VLSI chips are designed by computers, which also guide the laser beams that etch the circuits onto the silicon — a good example of a technology being used to supercede itself.

Many scientists are predicting, however, that semiconductor miniaturisation is approaching its limits, and that any further reductions in the size of electronic components will have to be based on radically different technology.

VON NEUMANN MACHINE

The concept of the *von Neumann machine* has profoundly influenced the development of computers ever since their inception in the 1950s. It was John von Neumann's theoretical architecture of a computer that became the blueprint for the engineers who actually built the first electronic computers.

The computer was divided by von Neumann into four main components: an arithmetic logic unit, a control unit, some kind of addressable memory and input/output facilities. Within von Neumann architecture, the program and data are both held in memory.

Around this architecture, the von Neumann machine works as follows. The first program instruction is located in memory, fetched to the processor, decoded and acted upon. In addition, all arithmetic and data manipulations are done by taking data to the processor and replacing it after it has been worked on. Once the instruction has been performed, a program counter will be loaded with the location of the next instruction to be acted on.

Clearly, the von Neumann machine closely resembles the actual workings of modern computers. The essential feature of the von Neumann machine is that it is sequential in nature — control is not passed to a particular instruction until its memory location is loaded into the program counter.

Although this model has served well for over 40 years, the sequential basis of the machine, particularly the bottleneck caused by directory operations through the processor, is proving to be a major barrier preventing the long-term development of artificial intelligence. Much research is now being put into a 'non-von Neumann architecture'.

The starting point for these machines is that the flow of control should be governed by the result of previous instructions, rather than depending on a program counter. Because the flow will not be sequential, the result could trigger the flow of external control in several directions. This would enable a number of parallel tasks to operate simultaneously, a major attribute of any machine claiming an artificial intelligence.

WILDCARD

The *wildcard* is a 'dummy' character that can be inserted into a filename. When the computer uses the filename to perform an operation, usually a search of some description, the computer will ignore the wildcard character when looking for a filename match. This process is aptly named 'fuzzy matching'.

There are two general instances in which wildcards are useful. When you want to manipulate a number of files that have similar names — say, Prog1, Prog2 and so on — the files can be targeted with a single command. Secondly, wildcards provide a useful tool for obtaining the data from databases when you are unsure of the exact name of the record or file you're looking for.

There are two characters most often used in providing wildcards on microcomputers – * and ?. The question mark character is used to replace single characters in a filename – an example of this is PROG?. BAS. If, however, we wish to refer to all the files with the BAS suffix, rather than typing five question marks, simply typing *. BAS would accomplish the task.

WINCHESTER DISK

A *Winchester disk* is a particular type of hard disk that was developed for the IBM 3340 disk drive by the company's laboratory in Winchester. The density of information that can be stored on a Winchester disk is 300 tracks per radial inch and 3600 bits per inch around any track. Winchester disks used with microcomputers commonly have a capacity of 20 Mbytes, while mainframe versions can hold several hundred Mbytes of data.

In the attainment of such high packing density, the manufacture of the disks leaves them extremely susceptible to damage. Manufacturers go to great lengths to protect Winchester disks, which are characterised by being sealed in airtight containers to protect them from dust. Extra light read/write heads are fitted to the drive mechanism, and the disks are coated with special materials to minimise wear and tear.

Blueprint Pioneer
Janos Louis von Neumann (1903-1957) is one of the key figures in the development of the computer. His theories on the architecture and design of electronic computers, incorporating a central processing unit and addressable memory, were used in the development of the first computers in the late 1940s and early 1950s

ASSOCIATED PRESS

THE WRITE STUFF

Making Mega Sales
Virgin's Games Centre in London's Oxford Street is typical of a new generation of outlets fuelled by the rise in popularity of board games and computer software. However, most computer software is still sold via the large retail chains, such as W H Smiths and Boots. Persuading a retail chain to stock your product is often difficult, particularly at peak sales seasons, but is mandatory if good sales figures are to be achieved

Home Office Star
Jeff Minter is well-known to many arcade games enthusiasts. His programs have been hugely successful in a highly competitive market, and all are developed at his home. The position of programmers like Minter can, however, be unstable in the long run. Commercial programmers can benefit from the training and interchange of ideas that comes from working for a large organisation, benefits often denied to their self-employed counterparts

For many micro users, a career in games programming represents a dream come true. There are, in fact, many fantastic aspects to the industry, but not all of them will net you a pot of gold. However, a careful consideration of your abilities and the marketplace might very well earn you a comfortable living.

Freelance programming is a risky business. First, payment is likely to be low — unless the game is a huge success. Top names may receive an advance and royalty percentage, but an unknown programmer is unlikely to receive anything in advance and only a small royalty. Secondly, it will be quite some time before any payment is received. Preparing a game for launch can take several months, and most royalty payments are paid six months in arrears, so you'd be well advised to expect about a year to pass from completing the program to receiving your first payment. And, of course, there's no guarantee that the game will be accepted — even if you have a contract. Any contract will have a clause stating that the game must be completed to the satisfaction of the software house, and it's a sad fact that some types of game go out of fashion very quickly. What was a popular concept when you began work on the program may well be as dated as Pacman by the time you complete it.

The majority of games software houses rely primarily on freelance programmers, but there are employment opportunities. If you have the talent and the ideas, most software houses will jump at the chance to employ you. The advantages of employment over self-employment are, in theory at least, security, promotion prospects, training and the feeling of being part of a team. In practice, security, promotion prospects and training are far from guaranteed in this business.

With software houses appearing and disappearing regularly, it's debatable whether there is any such thing as a secure job in games programming. Obtaining accurate statistics is difficult, since many 'software houses' are nothing more than a trading name and a garage or garden shed. In the UK there are probably around 300 to 350 games software houses in existence at any one time. About 200 or more are formed each year, and roughly the same number go out of business in the same period. This balancing out process gives a false impression of stability: although the total number of companies remains reasonably constant, the failure rate is probably in excess of 50 per cent of the size of the industry!

Training and promotion prospects depend very much on the individual company. A few of the larger companies provide formal training, but the vast majority do not. The typical software house organises programmers into teams, with a senior programmer as the team leader. Junior programmers are able to consult the team leader for help and advice. One or two companies pay for junior programmers to attend evening classes at local colleges, but normally only the course fee is met, so you attend in your own time.

In the smaller companies, promotion prospects are slim or non-existent. The managing director runs the company, and the programmers write the software. There may be a sales/marketing manager to sell the games, but that is normally the extent of the staff. In larger companies, the usual promotion ladder is from junior programmer to team leader, and perhaps on to management if the individual has the aptitude and the inclination. Promotion depends almost entirely on ability; if you are good, you can be promoted to team leader in six months. If you do not show the aptitude for

TONY SLEEP

the job, being with the company five years is unlikely to count.

Qualifications, and even experience, count for little. What employers want to see is evidence of your ability (that is, a completed game), fresh ideas and market awareness. Most software houses say that they prefer formal qualifications, but all agree that a lack of these would not stand in the way of a talented programmer. And many prefer a 17-year old with ideas to a 40-year old with experience.

You will, of course, need to program in machine code. Either Z80 or 6502 is usually required, and 68000 is becoming increasingly valuable. BASIC, unfortunately, won't get you anywhere. You also need in-depth knowledge of one of the currently popular home micros, say the Sinclair Spectrum, Commodore 64, BBC Micro and the Amstrad CPC range.

Games Programmer: Factsheet

Qualifications Required
Formal computing qualifications help, but they're usually not essential.

Languages Required
Machine code. Normally Z80, 6502 or 68000. You will also be expected to demonstrate in-depth familiarity with a popular home micro.

Salary Range
Extremely variable. A non-graduate would normally join a software house at anything between £4,000 and £9,000 per year, depending on ability and size of company. Graduates can expect a salary in line with normal industry rates.

Training Provided
Formal training is very rare. Most companies give informal, on-the-job training under a senior programmer.

Promotion Prospects
Virtually none in the smaller software houses. In larger ones, you can expect to progress to a senior programmer and possibly on to management on merit.

Selling Your Game

You've just written a game destined to become more famous than Pacman, Defender or even Space Invaders. How do you go about selling it? This quick check-list is designed to put you on the right track.

Protect Yourself
Most software houses are scrupulously honest. A few, however, are not. You can protect your game by taking a few simple precautions:

- Include a copyright notice in the first line of the code. If possible, use the tricks of your machine to make sure that it cannot be removed; otherwise, embed further copyright notices throughout the game.
- Include a few redundant lines: that is, lines that appear to be part of the game but which actually have no effect. If the game appears without your permission, you can use these redundant lines to help establish your copyright.
- Lodge a copy of the program with your bank, and get a dated receipt.
- Keep everything. Earlier versions of the program, flow-charts, bits of routines, scribbled notes — anything that will help you prove that you wrote it.
- If possible, take the program to the software house personally and get a signed and dated receipt. If you do use the post, send it recorded delivery.

Make It Complete
Software houses will happily attend to the presentation of the program. They may add on their own title screen, write the instruction leaflet, add in a fast loader and so on. They will, however, expect the game itself to be complete in every respect.

Pay Attention To Presentation
Software houses receive a large number of unsolicited submissions. Anything you can include to make yours stand out will help, such as:

- A brief 100 to 200 word outline of the game.
- Clear instructions for loading and playing.
- Several copies of the game in case of loading difficulties (disks are preferable to tape).
- A flowchart, if it helps to clarify the structure of the game.
- Details of any requirements over and above the basic machine. For example, 48Kbytes RAM, joysticks, a disk drive and so on.

Mark everything with your name, address and daytime telephone number.

Keep Your Options Open
Send your game to as many software houses as possible, and don't accept the first offer the instant it arrives. If you do receive an offer, ring a few of the others and let them know that you have been made an offer but don't give any details. That may speed up their assessment of your game.

Never sign a contract without professional advice. Talk to a solicitor. In Britain, many solicitors offer a fixed-fee interview of around 20 minutes for between £5 and £10. This is money well-spent.

TONY SLEEP

Experience Counts
Dave Nicholls works full-time as a programmer for the Ram-Jam Corporation. The bulk of his work consists of converting programs from one system to another. Nicholl's background as a VAX minicomputer programmer and his commercial experience make it easier for him to move from one system to another and to keep track of new developments

ALLOCATING RESOURCES

We complete our series with a look at the interface between C and the Unix operating system. Our discussion introduces some more useful functions from the C library (see page 1844), and we give an index compiler program that demonstrates many of the language's features in action.

One of the major uses of the C language is in writing systems software, operating systems and utilities, as well as efficient applications software. This means, among other things, that there must be a good interface between the language and the operating system under which it is running, and that it must be possible to program operating system functions.

One particularly useful function when writing code that is to be transferred between different systems is sizeof(object). An operator rather than a function, it can be provided with either a variable name or a type name, and returns the number of bytes used for storage of that object. One example of its use might be to write code that works whether int is 16 or 32 bits. It is often used in conjunction with four functions (calloc(), malloc(), realloc() and free()), which all deal with the allocation of memory.

- **calloc():**

```
char *calloc(number, size)
int number, size
```

The command calloc() allocates, and initialises to zero, an area of memory sufficient to hold number items, each of size bytes, returning a pointer to the first item (or NULL if insufficient memory is available). For example, to reserve space for 50 integers, we could use:

```
p=calloc(50, sizeof (int));
```

- **malloc():**

```
char *malloc(bytes)
unsigned bytes;
```

This performs a similar function but simply reserves bytes number of uninitialised bytes. With both of these functions, the value returned is simply a pointer to char, since char is always one byte. To actually use it as a pointer to an object of a different type it must be cast.

- **realloc():**

```
char *realloc(p, bytes)
char *p;
int bytes;
```

The realloc() command changes the size of the area pointed to by p to bytes number of bytes. It will copy the contents of the old area into the new area but it may or may not reallocate a new section of memory so pointers into the area will not be valid afterwards.

- **free():**

```
free(p)
char *p;
```

This function frees an area of memory that has been allocated by calloc, malloc or realloc for reuse. It could be a disastrous error to use it with any other pointer value.

Another class of C functions deals with the handling of errors. Most errors that can arise within a C program occur when handling input/output or on library calls. The library includes a standard int variable called errno and a list of system error messages sys_errlist[], which can be referred to directly from within the program. If an error occurs during a library call then a value will be placed in errno to indicate this fact. A call to:

```
perror(s)
char *s="my error message";
```

will display on stderr both the given and the system error message corresponding to the current value of errno.

When errors occur on input/output they will sometimes produce a normal system error, but this cannot always be relied on. The function int ferror(file_pointer) will test whether an error has occurred in reading or writing to the given file, returning zero if an error has occurred and non-zero otherwise. The error condition may prevent access to the file, so the function clearerr (file_pointer) is provided to reset the error indication.

A number of functions are also provided for general housekeeping of disk files and directories. Full details of these should be supplied with your compiler, but they will almost certainly include access(), which checks the access mode of a named file or directory; chmod(), which changes access mode; and chdir() for changing the current working directory.

The example program is fairly long and involves almost all the features of C we have looked at so far, including some (like recursive calls to functions) that we have not covered explicitly, as well as malloc, to get storage space to maintain a linked list. The object of the program is to create an index for a large text file — in other words, a list of all words used and the page numbers on which they appear. The words are kept in a linked list that grows as each word is encountered. Associated with each word is a pointer to another linked list of page numbers for each word.

C is a very interesting language to use and very widely available. It's one of the few languages with which it's quite possible to write a program on a small micro that will also run on large micros, minis and even mainframes. If you haven't already tried using C, it would definitely be worth the effort — its qualities are many and its popularity is growing in leaps and bounds.

Index Compiler

```
/* In file index.h */
#define NULL 0
#define MAXWORDSIZE 20
typedef char ENTRY [MAXWORDSIZE];
/* Each element in the index will consist of the
actual entry, a pointer to the list of page numbers
and a pointer to the next element */
struct page_number
{
    int pn;
    struct page_number *pnext;
}
typedef struct page_number page;
typedef page *plink;
struct index_element
{
    ENTRY entry;
    plink pages;
    struct index_element *exit;
}
typedef struct index_element element;
typedef element *link;
/* we can now refer to an item in the list as an
'element' and a pointer to an element is called a
'link' */
/* in file index.c */
#include stdio.h
#include string.h
#include index.h
#define LPP = 66; /* lines per page */
main(argc,argv)
int argc;
char *argv[];
link head;
{
    FILE infile;
    int lc = 0, pc = 1, inword = 0;
    ENTRY nextword;
    char *nw;
/* initialise list with a first dummy entry (it makes
life easier!) */
    head = new_entry ("",NULL, 0);
/* get file name to index */
    if(argc != 2)
    {
        fprintf (stderr, "\nusage is %s filename\n",
        *argv);
        exit(1);
    }
    if (infile = fopen (*++argv, "r") == NULL)
        fprintf(stderr, "\nfile not found %s\n",
        *argav);
         exit(1);
    }
    nw = nextword;
    while (c = getc(infile) != EOF)
    {
        if(c == '\n')
/* add one to line count and check for end of page */
        {
            lc += 1;
            if (lc>LPP)
            {
                pc += 1;
                lc = 0;
            }
        }
        else if (inword)
        {
            if (isalpha(c))
                *nw++ = c;
            else
            {
                inword = 0;
                *nw = '\o';
                insert (nextword, head, pc);
                nw = nextword;
            }
        }
        else
        {
            if (isalpha(c))
            {
                inword = 1;
                *nw++ = c;
            }
        }
    }
    if(inword)
    {
        *nw = '\o';
        insert (nextword, head, pc);
    }
    display_index;
}
insert(e)
ENTRY e;
{
insert(e,l,pnum)
ENTRY e;
link l;
int pnum;
static link lastl;
{
    lastl = head;
    if(l == NULL)
        {
            lastl->next = new_entry (e,l,pnum);
            return;
        }
    else
    {
        int s;
        s = strcmp(e,l->entry);
        if(s == 0)
/* word already present so add page number */
        {
            add_page_number (pnum,
            l->pages);
            return;
        }
        else if(s > 0)
/* gone too far so insert new node after last one in
list */
        {
            lastl->next = new_entry (e,l,pnum);
            return;
        }
        else
        {
/* not found yet so move on through the list using a
recursive call to insert */
            lastl = l;
            insert(e,l->next, pnum);
            return;
        }
    }
}
link new_entry(e,l,pnum)
ENTRY e;
link l;
int pnum;
{
/* get enough space for new entry using malloc */
    link newl;
    newl = (link) malloc (sizeof (element));
/* note cast to convert char pointer returned by
malloc */
    newl->entry = e;
    newl->next = l;
    newl->pages = (plink) malloc (sizeof (page));
    newl->pages->pn = pnum;
    newl->pages->pnext = NULL;
    return(newl);
}
add_page_number (pnum,pl)
int pnum;
plink pl;
{
/* find end of page number list */
    while (pl->pnext) != NULL)
        pl = pl->pnext;
    pl->pnext = (plink) malloc (sizeof (page));
    pl->pnext->pnext = NULL;
    pl->pnext->pn = pnum;
    return;
}
display_index ;
link l;
plink pl;
{
    l = head->next;
    while(l != NULL)
    {
        printf ("%s\t", l->entry) ;
        pl = l->pages;
        while (pl->next != NULL)
        {
            printf("%d4,",pl-> pn);
            pl = pl->next;
        }
        printf("%d\n", pl->pn);
    }
    return;
}
```

SOURCE MATERIAL

Machine code is the same regardless of the assembler used — but the choice of assembler affects the speed at which programs can be written and the quality of the final product. Here we look at what makes a good assembler and examine three popular packages for home machines.

Although an assembler is essential for the serious programmer, it's only one of the many tools required. The assembler converts the program's original *source code* (written in the mnemonics and symbols of assembly language) to *object code* (the actual machine code). A monitor or debugger program is also necessary to actually test the program and locate its faults.

Most assemblers follow the assembly language defined by the CPU manufacturer, but small or simple assemblers often use non-standard syntax to make their jobs easier. To load an immediate value into the A register of a 6502, for example, the correct assembly language is LDA #$20, whereas a simple assembler might use LDAIM $20. Non-standard assemblers should be avoided, however, since they can be difficult to learn and, consequently, you will have to convert existing source files before they will assemble.

Hisoft Devpac

For the Amstrad CPC464, 664 and 6128, Spectrum 48K, CP/M and MSX micros

Hisoft's popular Devpac is a complete development tool with assembler, editor and machine code debugger. Programs are developed using numbered lines similar to BASIC, but it also has full editing features — renumber, delete and move lines, find and replace and so on. The Devpac is a two pass type with a wide range of directives and comprehensive arithmetic abilities. A wide range of assembly options gives the programmer flexible control of the way the code is produced.

Both the assembler/editor program and the debugger are relocatable so that you can load them into memory at a suitable point for the program you are developing. If both programs are loaded at once, they will be linked, enabling you to jump easily from the assembler to the debugger when necessary. The debugger mimics the 'front-panel' found on mainframe computers by producing a near-full screen display showing the contents of a block of memory, the CPU registers and the instruction about to be executed. As well as the usual facilities, Hisoft provides breakpoints, single-stepping and trace features, making this an ideal tool for debugging programs. Its comprehensive screen display is particularly useful for beginners.

Devpac comes with a comprehensive 32-page manual. Although neither the manual nor the Devpac commands themselves are as clear as they might be, there is little information omitted that the programmer might need to know. Devpac is a comprehensive and well-produced package

Assembler Type
Two pass, standard mnemonics

Limits
Source Code: Limited by RAM but file linking is available
Symbol Table Size: User specified

Assembler Directives

ORG	Set origin of program
EQU	Assign value to symbol
DEFB	Store 8-bit value(s)
DEFW	Store 16-bit value(s)
DEFS	Reserve storage space
DEFM	Store ASCII string
ENT	Marks program start address for assembler RUN command
END	Marks end of source program
IF. . . ELSE	Conditional assembly
*E	Eject a page on printer
*H	Defines heading on listing
*S	Pause listing
*L	Turn listing on and off
*D	Give addresses in listing in decimal
*F	Include pre-written source file at this point
*T	Write object code to tape instead of memory

Assembly Options
Produce symbol table or not
Generate object code or not
Produce assembly listing or not
Place object code after symbol table or at ORG address
Check that object code does not overwrite Devpac or don't check

Arithmetic
+, -, *, /, MOD, AND, OR, NOT and XOR. Decimal, hex and binary

Price
Amstrad £21.95; Spectrum £14.00; MSX £19.95, CP/M £39;95
Hisoft (0582) 696421

The simplest of assemblers are built into debugger programs. Termed 'line assemblers', they allow one line of assembly language to be entered and assembled directly into memory. However, they do not allow the programmer to assign labels and symbols to the addresses and values used in the program, making them suitable only for making short corrections to programs already in memory.

Debuggers often include facilities such as inserting breakpoints, single-stepping and tracing the execution of a program. Insertion of a breakpoint causes a jump instruction back to the debugger that will be inserted in the code and automatically removed after the break has been reached. The status of registers and memory locations can thus be checked at any point.

Full assemblers are described as being either 'single pass' or 'two pass'. The simpler single pass assembler reads through the source code only once, so if the program jumps forward to a labelled instruction, the assembler will not know the address that the label refers to. In this case, the assembler will go back and put the address in once it has discovered it; however, the actual address will not appear in the assembly listing. A two pass assembler is more commonly used — it reads through your source program once, checking its syntax and building a complete list of the symbols used in the program before reading through it again and producing a completed assembly listing.

Every assembler has its own set of 'directives', or 'pseudo-ops'. These are instructions that are not part of the CPU instruction set and are not translated into operation codes. Rather, they tell the assembler to perform some specific task, such as turning the printer on, or reserving a block of memory space within the code for data storage. Directives vary from package to package, both in what they do and in their names and syntax. Most are reasonably standardised, but unusual directives can be a problem if you are trying to get a program written for one assembler to work with another.

The first directive you'll probably come across is ORG, which tells the assembler where the program to be assembled will be loaded in memory. All assemblers also have a way of defining symbols — COUNT EQU 5, for example, would set the symbol COUNT to 5. Additionally, a good set of 'data directives', which are instructions that allow you to set up the data storage areas of your programs, is helpful. There are usually instructions to store particular values (DEFB, DEFW) or strings (DEFM), and to free some bytes for the program (DEFS).

ARITHMETIC OPERATIONS

Most assemblers are capable of arithmetic, and it's helpful if they provide a full range of operators: +, -, /, *, AND, OR, NOT and MOD. The last, MOD (or an equivlent), is virtually essential, since programmers frequently need the assembler to split a 16-bit value into two eight-bit values. Good assemblers are also capable of handling numbers

Picturesque Editor/Assembler

For the 16K and 48K Spectrum

Picturesque's Editor/Assembler has long been a favourite of Spectrum programmers, both hobbyists and professionals alike. It provides a full two pass Z80 assembler and editor on the Spectrum and can be optionally used with Picturesque's Monitor package. It operates with both 16K and 48K machines, although its facilities are more restricted in the 16K version. To alleviate the problems in showing wide assembly listings on the Spectrum's 32-column display, the program simulates a 40-column screen.

The 48K version has the very important ability of assembling from source code files stored on tape or Microdrive. This effectively accommodates source programs of up to 95 Kbytes in length and, therefore, the creation of large object programs, such as those necessary for commercial games. This is a standard facility on assemblers for business micros. Many home computer assemblers are limited to programs short enough for both their source and object code to be in memory at once.

Picturesque supplies an excellent 56-page manual for the Editor/Assembler, which contains clear and comprehensive information on all aspects of the package. Although there are more recent alternatives that boast more assembly features, Picturesque is an excellent yardstick and a capable package in its own right.

Assembler Type
Two pass, standard mnemonics

Limits
Source Code: Up to 95K using Microdrives
Symbol Table Size: Limited by free memory

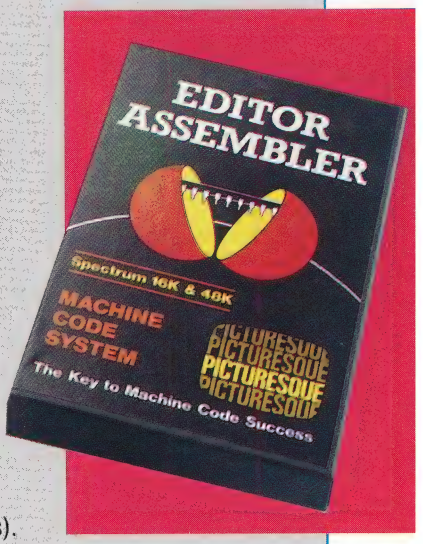
Assembler Directives

ORG	Set origin of program
EQU,	Assign value to symbol
DEFL	As EQU but symbol may be reassigned several times
DEFB	Store 8-bit value(s)
DEFW	Store 16-bit value(s)
DEFS	Reserve storage area
DEFM	Store ASCII string
PRNT	Turn printer on and off
END	Marks end of source program

Assembly Options
Assemble from memory, tape or Microdrive
Assembly listing on printer or screen

Arithmetic
+, -, < and > (high and low bytes of 16-bit values).
Hex and decimal

Price
Editor/Assembler £8.50; Monitor £7.50
Picturesque (01) 777 0372



in decimal, hex and binary, as well as being able to evaluate an ASCII character (so, for example, you could use A instead of having to look up its value yourself in an ASCII table).

When you ask for your program to be assembled, you are usually offered the chance of specifying some options as to how the assembly is performed. These can include whether a listing and symbol table is produced, and whether or not it is displayed on screen, sent to the printer or written to a disk or tape file. One very useful facility is a trial assembly, whereby the program is assembled but no machine code is loaded into memory (this is a quick way to check that the program assembles without actually doing it!).

The best assembler packages allow both conditional and macro assembly. Conditional directives allow sections of listing to be assembled only if certain conditions are true. This means that one source listing can generate several versions of the object code depending on the requirements. A macro allows you to define a new assembly language mnemonic in terms of several existing ones. When the macro is assembled, the assembler automatically substitutes the correct instructions. By building up a library of macros to do particular jobs (such as printing messages, opening data files and so on), it's possible to write programs very quickly. However, they are not as efficient as hand-coded programs, in execution speed and size.

Besides the important directives, the other fundamental feature of assemblers is how they are organised. Many store both the source code and the object code in memory along with the assembler/editor, which limits the maximum size of program you can write. A solution involves being able to assemble from a source program in storage, which is usually practical only on machines with disks or fast tapes, such as the Sinclair Microdrive. But this is the only way to create large programs.

Besides the assembler's specifications, you should check how fast it is. A program will be assembled many times during the course of its development, so a small fast assembler is often preferable to a slower one with many features. The quality of editor provided is also important. The ideal is a full screen editor similar to a word processor, although many programs rely on line editors.

Watford ROMAS

BBC Model B

ROMAS is a professional 'cross assembler' for the BBC B. It's designed to allow programs to be developed not just for the BBC but for a wide range of other microcomputers. The PROC directive is used to tell the assembler which CPU it is assembling for. ROMAS currently supports the 6502, 6511, 8085, Z80, Z8, 6809, 65C02, 8041 and 8048 processors, making it suitable for almost all eight-bit computers and microprocessor-controlled devices. ROMAS includes utilities to allow object programs to be trasmitted to other micros and EPROM programmers.

Besides being a cross assembler, ROMAS supports conditional assembly and has a range or directives designed to help maintain large programs and produce versions of them to run on different machines. For example, the TYPE and INPUT directives allow the assembler to request from the user values for symbols during assembly. In conjunction with the conditionals, it's possible for a program, when assembled, to ask 'which machine? 1 = Spectrum, 2 = Amstrad, 3 = MSX' and then produce the correct version of the program. ROMAS assembles from a disk file and writes the object code back to a disk file, which will subsequently be tested on the BBC, a BBC second processor or the micro for which it was designed.

The package includes a screen editor called EDT, which supports macro commands; complex operations can be built up in terms of existing commands. ROMAS is undoubtedly one of the most sophisticated editors available on a small micro, including a 156-page manual and a sample source program (a BBC disassembler). Although most users will be content with the assembler built into BBC BASIC with the addition of a monitor program, ROMAS is a professional alternative for serious software developers

LIZ HEANEY

Assembly Type

Two pass cross assembler, standard mnemonics

Limits

Source Code: Limited by disk capacity
Symbol Table Size: 7K, but can be extended

Assembly Directives

ORG	Set program origin
EQU	Assign value to symbol
DB, DW, DS	Store 8 and 16-bit values, ASCII strings and reserve space
IF, ELSE, ENDIF	Conditional assembly
END	Marks end of program
EXTEND	Continue assembling from named source file
HIMEM, LOMEM, MODE	As in BBC BASIC
TYPE, INPUT, GET	Interact with user during assembly
TITLE, PAGE, EJECT, LIST, TAB, WIDTH	Listing controls
PROC	Select processor to assemble for

Assembly Options

Produce assembly listing either on screen or printer

Arithmetic

*, /, REM (modulo), +, -, AND, OR, NOT, XOR, shift-left and shift-right. Decimal, hex, octal and binary

Price

£45
Watford Electronics (0923) 40588

INTERRUPTED SERVICE

In the previous instalment of our series on the Motorola 68000, we looked at the basic serial input/output mechanisms, using the ACIA chip as an example. We'll now extend the I/O to parallel interfaces and give an introduction to interrupt handling.

Let's begin by looking at an example of serial output with a subroutine called message. This routine has the address of the message passed to it using the A3 address register. So, for example:

```
LEA   TEXT,A3    set up pointer to the message
                 TEXT
JSR   MESSAGE    output the message
```

would output the message TEXT stored as:

```
TEXT: DC.B       'Home Computer',$00
```

where DC.B is an assembler directive to declare memory space for the text 'Home Computer'. The blank byte $00 is just a message terminator, informing the message subroutine that there is no more text.

The subroutine to output all the bytes in the array TEXT would use the subroutine OUTCH, which we looked at in the previous instalment (see page 1858). In that instance, OUTCH used the 'ready' bit to output each character as soon as the ACIA was ready to send another character. So the message subroutine would look like:

```
MESSAGE: MOVE.B   (A3)+,D0   get next message
                             byte
         BEQ      DONE       check for end of
                             message
         JSR      OUTCH      output the
                             character
         BRA      MESSAGE    loop until end of
                             the message
DONE:    RTS
```

Each byte of the message is copied into D0 (the data register used as a value parameter for OUTCH), using A3 as a pointer with post-increment indirect addressing. If the byte was zero, exit would be made via the label DONE; otherwise, the character is output using OUTCH.

If we require output data in a parallel mode, where all the data bits appear simultaneously, then the PIA chip provides the facilities. However, since this chip is a general-purpose interface chip, we need to configure it to our own particular hardware configuration. This is much along the lines of the ACIA chip, where we have to set bit rates and byte formats.

In the case of the PIA chip, we have to configure the eight parallel lines of *both* halves of the chip (A and B) to be either input or output lines — that is, we have to configure the data direction. To do this we write to the control register (CRA or CRB) bit 2 to indicate we wish to set the data direction in a data direction register (DDRA or DDRB). For example:

```
CONFIGPIA: CLR.B    PIACRB       set control
                                 register bit
                                 to zero
           MOVE.B   #$FF,PIADDRB set data
                                 direction
                                 for output
           BSET     #2,PIACRB    revert to
                                 normal
                                 data
                                 register
           CLR.B    PIACRA       address
                                 data
                                 direction
                                 for A
           CLR.B    PIADDRA      set data
                                 direction to
                                 input
           BSET     #2,PIACRA    revert to
                                 normal
                                 data
                                 register
```

would set all of side A to input, and side B to output. Actual data transfers to the parallel devices would then be accomplished with:

```
MOVE.B   D0,PIADRA    output contents of D0
```

and for input we would have:

```
MOVE.B   PIADRA,D1    read input into D1
```

Note that all the PIA addresses just referred to would be set up initially, as in:

```
PIADRA   EQU   $30051
PIACRA   EQU   $30053
```

Notice that the significance of the data read from, or written to, the PIA depends on the type of device that is electrically connected to the digital channels of the chip — for example, we could have a seven-segment display connected (as shown in the diagram) from which you can see that any decimal digit can be constructed using the appropriate segment. So, for example, to display the number 3, we will have to energise segments 1, 4, 2, 5 and 3.

Not only would you need to transmit data to the peripheral device but you might also need to control the electrical function of it. So in the case of the seven-segment display, you might need to latch the data into the device using some of the

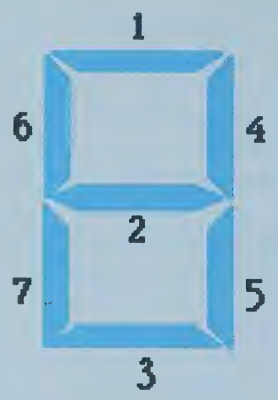
spare bits on the data word written out. Very often these control signals simulate an electrical 'clock' pulse, and would therefore involve the setting and resetting of the control bit. So:

```
BSET    #CONBITNO,PIADRA
                    set the control bit
JSR     DELAY       wait for a very small delay
BCLR    #CONBITNO,PIADRA
                    and reset it after
```

would provide the 'clock' on whatever digital channel is assigned to CONBITNO on address PIADRA.

Parallel Motion

The diagram below shows the structure of a typical PIA chip, used to provide parallel input/output facilities for the 68000. The chip provides two ports, which are conventionally assigned — one is for input and the other for output. Each port has its own control, data and data direction registers. A typical application might be to drive a seven-segment LED display of the type shown on the left



CAROLINE CLAYTON

INTERRUPTS

Although the subject of interrupts is often misunderstood, the objectives of using interrupts in a computer system are really just concerned with the efficient use of the CPU and being able to respond to external events. We do not want, for example, to waste CPU time while characters are being printed, as in the example of the OUTCH subroutine. Also, in order that we can do something else, we need to know when the printing has finished so that we can output the next available character.

The situation is even worse if the program is waiting for input; say, from a keyboard. The efficiency of the system depends, of course, on the speed of typing and also on the other tasks being undertaken by the CPU, such as outputting to the printing in parallel with waiting for keyboard input.

In order to achieve this parallel operation, we must organise the logical sequence of events within the machine to make sure we do not lose control of the program or any data. Let's look at what is therefore required.

- **Save the computer state.** We need to do this so that we can return to the program that we have interrupted without any noticeable effect or loss of data (we will, of course, lose time inevitably). But we must first define what constitutes the 'state' of the computer.

We could consider this to be the entire program and data area of the user program with its register state and the program counter. In practice, since we do not (in general) expect any changes to the user program from the interrupt source, it is sufficient to save just the program counter (PC) and the status register (SR). This should then give us a complete indication of the interrupted program's state.

- **Locate the interrupt source.** This is required because there is probably more than one peripheral device connected to the computer and we need to know which 'service' routine to enter.
- **Inhibit interrupts from other sources.** We may need to do this because if another interrupt arrives while we are servicing an existing interrupt, we could lose data. But this could only occur if the time taken to service the second interrupt was greater than the time between the arrival of data on the first interrupt.
- **Enter the interrupt service routine.** We need a mechanism for doing this — either automatically or by executing a specific instruction.

• **Return to the interrupted program.** Finally, after the interrupt has been serviced, we need to return to the exact state of the program just prior to the interrupt.

Let's see how this logical organisation is achieved on the 68000. First of all, the state of the machine is saved using the stack to store first the user's PC (as a full long word) and then the SR. With this mechanism, interrupts may be nested so that we can interrupt an interrupt routine.

If the interrupt routine is going to use some of the registers that have already been allocated to a specific purpose, then the routine can save those registers on entry and restore them on exit. The MOVEM instruction would do this very easily for us on the 68000:

```
MOVEM   D1,D3,-(SP)     stack the data
                        registers
        |               interrupt service
                        code
MOVEM   (SP)+,D1,D3     and restore them
```

The problem of locating the source of the interrupt is easy on the 68000 because each interrupt is usually given a unique location in memory called a 'vector', which may be considered to contain a pointer to the address of the service routine. Of course, there may be several devices on a particular vector, in which case some device polling may be necessary in order to establish the interrupting device. However, this will not normally be the case on the 68000.

The need to arbitrate between interrupting devices is also taken care of for you. This is because the 68000 takes on the hardware priority of the interrupting source — that is, *only* interrupts of a higher priority will get the attention of the CPU. This means that high-speed data sources should be at a relatively high priority in a multi-interrupt system, to ensure that no data is lost.

Once the interrupt has the attention of the CPU, the PC is loaded with the contents of the interrupt vector, and the interrupt service routine is entered. The device is serviced either by loading the input data into a register, and from there to some sort of buffer, or by removing data from a buffer into the output device.

The only remaining action now is to return from the service routine to the interrupted program. This is achieved in much the same fashion as a return from a subroutine, this time using an RTE instruction rather than an RTS instruction. The RTE instruction reloads the PC and SR from the system stack automatically, so all interrupt service routines must end with this instruction. It also means that at that time, A7 (the system stack pointer) must be pointing at the interrupted program's registers. So if you pop data onto the system stack, make sure it's popped off before you execute the RTE instruction! Or, perhaps more appropriately, use another stack, since on the 68000 we can use any of the address registers as stacks.

SERVICE ROUTINE

Let's now look at a typical interrupt service routine in which we also have to poll devices in order to find the interrupting source. Within our theoretical system, we have two possible interrupt sources on what is called level 4 (a hardware interrupt line of priority 4). The two devices are an external stand-alone keyboard and a real-time clock. The clock is used to count seconds, which is used by other parts of the (software) system.

So that the interrupt vector for level 4 (at address $70) is set correctly, we need to initialise this when the program is first started, as in:

```
MOVE.L  # EXTDEVS,$70   set level 4 vector
```

Of course, there would be other registers to initialise as well, notably the user stack and the system stack. For example, we might use:

```
MOVE.L  # STACK,SP          set system stack
MOVE.L  # USERSTACK,A0      and the user stack
```

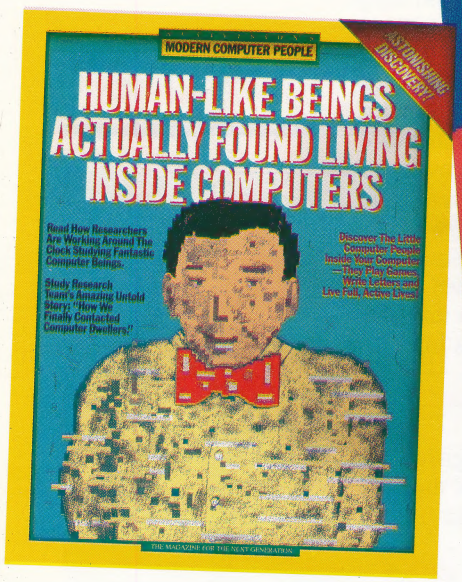
where we set A7 to the address STACK and A0 to address USERSTACK.

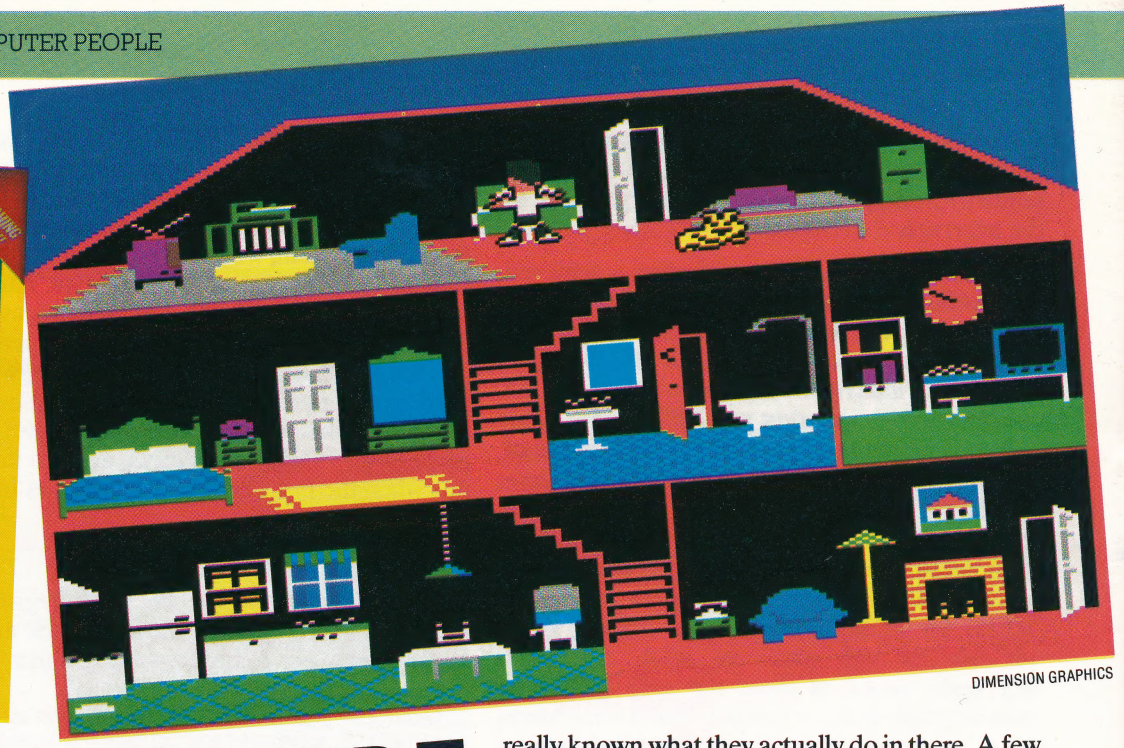
The interrupt service routine EXTDEVS would then be, for example:

```
EXTDEVS MOVEM.L  D0-D7,(A0)  save the data
                             registers
        BTST     # 7,PIACRA
                             see if the
                             keyboard
                             interrupted
        BNE      CHAR
CKCLK   BTST     #6,PIACRB   else it should
                             be the clock
        BNE      CLOCK
        BRA      WILD        if not then
                             spurious
CHAR    JSR      KEYBOARD    enter
                             keyboard
                             service
        BRA      CKCLK       but clock may
                             also have
                             interrupted!
CLOCK   JSR      SECS        enter clock
                             service
WILD
        MOVEM.L  (SP)+,D0-D7 restore the
                             data
                             registers
        RTE
```

In this routine, all the data registers are saved, because either CLOCK or CHAR will use them. Notice also that we poll the two devices by looking for specific status bits in the PIA control registers. We must also check the clock status after the keyboard has been serviced, since it is possible to have two interrupts together.

In the final article of this series, we'll look at the assembler and all the facilities that it offers. If used properly, it is perhaps the most powerful development tool available to the programmer.





DIMENSION GRAPHICS

COMMODORE PETS

A Day In The Life
The developers of the Little Computer People have gone to great lengths to make the LCPs appear as 'real' as possible. The magazine-style documentation provided with the program is some of the most polished to appear with any game. The house in which the action takes place is fully equipped to make life as comfortable as possible. Although, on the face of it, not a lot happens in the game, taking care of the 'pet person' has an emotional appeal that few other software packages can match

While not really fitting into any category of games software, Activision's Little Computer People for the Commodore 64 could nevertheless find itself at the top of the charts. We look at this unusual program, which revolves around keeping a tiny person happy who is living inside your computer.

Despite the fact that computer games software has developed considerably since the early days of home micros, most games still have a predetermined object to them. Whether arcade, adventure or strategy games, you'll still have to meet the various challenges set by the game's programmer. In so far as there is no definite end to the game, the Little Computer People is different — the 'person' within the game is intended to be a friend of the player.

The program has been written so that the person in each copy is to a certain extent unique. They all have different names, dress differently and have unique priorities in the way they 'live'.

There has obviously been a great deal of effort put into providing the little computer people (LCP) with histories to which you can relate. The software comes packaged with a large folder detailing the 'discovery' of the little people, their habits, preferences and habitats.

When the program is loaded for the first time, the screen displays the interior of a house, containing all the rooms you'd expect to see in a normal, three-storied home. There's also a closet in the bedroom that the person will often disappear into for a few minutes, although it's not really known what they actually do in there. A few minutes after loading, the LCP will enter the house and inspect the premises. If he approves, he'll disappear for a few moments before reappearing with all his belongings, including his pet dog.

To keep this person happy, you have to provide food, water and, above all, plenty of entertainment. Your success in satisfying these needs is judged by the person's facial expression, ranging from miserable to very happy. If the person looks unhappy and green, for example, then he's not being fed very well. You should keep in mind that the dog also needs to be fed.

Activision insists that the language spoken by these people has not yet been deciphered, so all communication with them is via the keyboard. In this way, it's possible to ensure that the person remains happy by suggesting he has a meal, lights a fire or plays the piano. An accurate barometer of the person's mood can be obtained if you suggest he writes a letter to you.

There is much attention to detail in the graphics and sound programming, which is of a very high quality. The music produced when the little person plays the piano is some of the best that has ever emanated from the Commodore 64. What's more, the LCP's hands have been synchronised with the notes and chords.

Although the idea of looking after a little person living inside your computer sounds absurd, the resulting efforts in keeping the person happy can be as addictive as any game. Even some of the most hardened games players have been known to develop an affection for the little creatures.

Little Computer People: For the Commodore 64
Price: £9.99 cassette, £14.99 disk
Publishers: Activision Inc, Box 7287 Mountain View, California, USA
Authors: Richard Gold, David Crane and Sam Nelson
Format: Cassette or disk
Joysticks: Not required

Motorola 68000 Instruction Set

Here, courtesy of Motorola Inc, we present the final instalment in a series in which we have given details of the 68000's instruction set broken into its eight component classes

Binary Coded Decimal Operations

Instruction	Operand Size	Operation
ABCD	8	$Dx_{10} + Dy_{10} + X \rightarrow Dx$ $-(Ax)_{10} + -(Ay)_{10} + x \rightarrow (Ax)$
SBCD	8	$Dx_{10} - Dy_{10} - X \rightarrow Dx$ $-(Ax)_{10} - -(Ay)_{10} - X \rightarrow (Ax)$
NBCD	8	$0 - (EA)_{10} - X \rightarrow (EA)$

NOTE -( ) = indirect with predecrement

Program Control Operations

Instruction	Operation
Conditional	
Bcc	Branch Conditionally (14 Conditions) 8- and 16-Bit Displacement
DBcc	Test Condition, Decrement, and Branch 16-Bit Displacement
Scc	Set Byte Conditionally (16 Conditions)
Unconditional	
BRA	Branch Always 8- and 16-Bit Displacement
BSR	Branch to Subroutine 8- and 16-Bit Displacement
JMP	Jump
JSR	Jump to Subroutine
Returns	
RTR	Return and Restore Condition Codes
RTS	Return from Subroutine

System Control Operations

Instruction	Operation
Privileged	
ANDI to SR	Logical AND to Status Register
EORI to SR	Logical EOR to Status Register
MOVE EA to SR	Load New Status Register
MOVE USP	Move User Stack Pointer
ORI to SR	Logical OR to Status Register
RESET	Reset External Devices
RTE	Return from Exception
STOP	Stop Program Execution
Trap Generating	
CHK	Check Data Register Against Upper Bounds
TRAP	Trap
TRAPV	Trap on Overflow
Status Register	
ANDI to CCR	Logical AND to Condition Codes
EORI to CCR	Logical EOR to Condition Codes
MOVE EA to CCR	Load New Condition Codes
MOVE SR to EA	Store Status Register
ORI to CCR	Logical OR to Condition Codes